CS434/534: Topics in Network Systems

Datacenter Resource Scheduling:
YARN, DRF, Mesos, Omega

Yang (Richard) Yang
Computer Science Department
Yale University
208A Watson
Email: yry@cs.yale.edu

http://zoo.cs.yale.edu/classes/cs434/

Acknowledgement: slides contain content from conference presentations by authors of YARN, DRF, Mesos and Omega.
Outline

- Admin and recap
- Cloud data center (CDC) applications/services
  - Fine-grained dataflow programming (e.g., Web apps)
  - Coarse-grained dataflow (e.g., data analytics)
  - Distributed machine learning using parameter server
  - DC resource scheduling
    - overview
    - YARN architecture
    - resource scheduling, DRF
    - alternative design: Mesos
    - alternative design: Omega
Admin

- Project meetings
  - Thursday: 2:00-4:00
  - Sunday: 3:00-4:30
Recap: DC Programming Frameworks

- **Noria**

```java
/* base tables */
CREATE TABLE stories
(id int, author int, title text, url text);
CREATE TABLE votes (user int, story_id int);
CREATE TABLE users (id int, username text);
/* internal view: vote count per story */
CREATE INTERNAL VIEW VoteCount AS
SELECT story_id, COUNT(*) AS vcount
FROM votes GROUP BY story_id;
/* external view: story details */
CREATE VIEW StoriesWithVC AS
SELECT id, author, title, url, vcount
FROM stories
JOIN VoteCount ON VoteCount.story_id = stories.id
WHERE stories.id = ?;
```

- **MapReduce**

```java
Map(input_key, input_value) {
    foreach word w in input_value:
        emit(w, 1);
}

Reduce(output_key, intermediate_vals) {
    set count = 0;
    foreach v in intermediate_vals:
        count += v;
    emit(output_key, count);
}
```
Recap: DC Programming Frameworks

- **Spark**

```java
val points = spark.textFile(...) .map(parsePoint).persist() 
var w = // random initial vector 
for (i = 1 to ITERATIONS) {
  val gradient = points.map { p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a, b) => a+b)
  w = gradient
}
```

<table>
<thead>
<tr>
<th>Transformations</th>
<th>java</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T → U)</code></td>
<td>RDD[T] → RDD[U]</td>
</tr>
<tr>
<td><code>filter(f : T → Boolean)</code></td>
<td>RDD[T] → RDD[T]</td>
</tr>
<tr>
<td><code>flatMap(f : T → Seq[U])</code></td>
<td>RDD[T] → RDD[Seq[U]]</td>
</tr>
<tr>
<td><code>sample(fraction : Float)</code></td>
<td>RDD[T] → RDD[T] (Deterministic sampling)</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td>RDD[K, V] → RDD[(K, Seq[V])]</td>
</tr>
<tr>
<td><code>reduceByKey(f : (V, V) → V)</code></td>
<td>RDD[(K, V), RDD[(K, V)]] → RDD[K, V]</td>
</tr>
<tr>
<td><code>union()</code></td>
<td>RDD[T], RDD[T] → RDD[T]</td>
</tr>
<tr>
<td><code>join()</code></td>
<td>RDD[(K, V)], RDD[(K, W)] → RDD[(K, V, W)]</td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td>RDD[(K, V)], RDD[(K, W)] → RDD[(K, Seq[V], Seq[W])]</td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td>RDD[T], RDD[U] → RDD[(T, U)]</td>
</tr>
<tr>
<td><code>mapValues(f : V → W)</code></td>
<td>RDD[(K, V)] → RDD[(K, W)] (Preserves partitioning)</td>
</tr>
<tr>
<td><code>sortBy(Comparator[K])</code></td>
<td>RDD[(K, V)] → RDD[(K, V)]</td>
</tr>
<tr>
<td><code>partitionBy(p : Partitioner[K])</code></td>
<td>RDD[(K, V)] → RDD[K, V]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actions</th>
<th>java</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td>RDD[T] → Long</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>RDD[T] → Seq[T]</td>
</tr>
<tr>
<td><code>reduce(f : (T, T) → T)</code></td>
<td>RDD[T] → T</td>
</tr>
<tr>
<td><code>lookup(k : K)</code></td>
<td>RDD[(K, V)] → Seq[V] (On hash/range partitioned)</td>
</tr>
<tr>
<td><code>save(path : String)</code></td>
<td>Outputs RDD to a storage system, e.g., HDFS</td>
</tr>
</tbody>
</table>

- **Parameter server**
Outline

- Admin and recap
- Cloud data center (CDC) applications/services
  - Fine-grained dataflow programming (e.g., Web apps)
  - Coarse-grained dataflow (e.g., data analytics)
  - Distributed machine learning using parameter server
  - DC resource scheduling
    - overview
    - YARN architecture
    - resource scheduling, DRF
    - alternative design: Mesos
    - alternative design: Omega
Problem Setting

- The ability to host multiple jobs, of the same framework or different frameworks (e.g., MR, Spark, MPI), achieving fine-grained sharing of the same cluster
  - Benefits of sharing infrastructure?
Coarse-Grained vs Fine-Grained Sharing

Coarse-Grained Sharing (HPC):
- Job/Framework 1
- Job/Framework 2
- Job/Framework 3
- Storage System (e.g. HDFS)

Fine-Grained Sharing:
- Fm. 1
- Fm. 2
- Fm. 3
- Storage System (e.g. HDFS)

+ Improved utilization, agility, sharing of data
Problem Setting

- Discussion: Key requirements of a generic, shared cluster?
Key Requirements We Focus on

- Extensibility/reusability
  - Allow different types of jobs; reduce duplicate efforts

- Scalability and robustness
  - Allow a huge number of concurrent jobs running at the same time

- Fairness/isolation
  - How resources are allocated to jobs can be controlled.
Outline

- Admin and recap
- Cloud data center (CDC) applications/services
  - Fine-grained dataflow programming (e.g., Web apps)
  - Coarse-grained dataflow (e.g., data analytics)
  - Distributed machine learning using parameter server
  - DC resource scheduling
    - overview
    - YARN DC control architecture
Architecture: Hadoop 1 (Monolithic Controller)

Discussion: extensibility/scalability issue?
Discussion

- How may you decompose the system to improve extensibility/scalability?
Architecture: Hadoop 2 (YARN): A Two-Layer Architecture
Architecture Component: ResourceManager

- Dedicated node, only one per cluster, running **centralized** scheduler

- **Functions**
  - Tracks resource usage, node liveness
  - Allocates resources to applications
    - pluggable scheduling policies (what is the scheduling complexity?)
  - Requests resources back from applications
Architecture Component: ApplicationMaster

- One per application
  - application-type specific
    - Runs on a worker node - must handle its own failures

- Functions
  - AM→RM: Requests resources, can specify number of containers, resources per container, locality preferences, priority
    - Can subsequently update requests with new requirements
  - Manages all scheduling/execution, fault tolerance, etc. of the same application (e.g., a MapReduce job)
Architecture Component: NodeManager (NM)

- Daemon on each node
- Functions
  - Monitor node resources faults
    - NM->RM: Report resource status
  - Manage containers
    - AM->NM: Container Launch Context (CLC) to NM to start container
    - Provide services to containers e.g. log aggregation
Discussion

- What you like about the control YARN architecture?
- What you do not like (or think missing) of the YARN architecture?
Outline

- Admin and recap
- Cloud data center (CDC) applications/services
  - Fine-grained dataflow programming (e.g., Web apps)
  - Coarse-grained dataflow (e.g., data analytics)
  - Distributed machine learning using parameter server
  - DC resource scheduling
    - overview
    - YARN architecture
    - resource scheduling
Isolation/fairness Approach 1: Static Partitioning

Problem: inefficient utilization.
Isolation/fairness Approach 2: Allocate Resources According to User Requests

- Assume allocating according to request, scaling to satisfy total demand
  - Users request 1, 2, 3, 4;
  - Total capacity 5;
  - Scale requests to allocate 0.5, 1, 1.5, 2 respectively

- Potential issue:
  - Mechanism allows user (gaming the system) to request higher than needed (4->10, will lead to a much higher allocation)
Simple algorithm

- Assume n users requesting R unit of resources
  - Compute R/n
  - Foreach user whose request $r_i \leq R/n$
    - Satisfy the request, remove $r_i$ from request (n--), $R -= r_i$
  - Allocate the remaining requests
Exercise

- Totally 15 units of resources
- Users requests: 1, 2, 3, 4, 8
- How are resources allocated by the alg?
Exercise:

Assume 2 users requesting \( R \) units of resources
Assume user 1 needs \( R_1 \) units of resources
Can user 1 improve its utility by requesting more than \( R_1 \) (not truthful)?

- Case 1: \( R_1 \leq R/2 \)
  - No need to lie (truthful); will get \( R_1 \)
- Case 2: \( R_1 > R/2 \)
  - If \( r_2 < R/2 \)
    » Get \( R-r_2 \)
  - If \( r_2 \geq R/2 \)
    » Get \( R/2 \)
  - Both independent of user 1 request (truthful is the best)
Problem Applying Fairness to DC Scheduling: Multiple Resources

Multi-resource example

- 2 resources: CPUs & memory
- User 1 wants <1 CPU, 4 GB> per task
- User 2 wants <3 CPU, 1 GB> per task
- What is a fair allocation?
Problem Applying Max-min Fairness in DC Scheduling: Multiple Resources

Most tasks need ~ <2 CPU, 2 GB RAM>

Some tasks are CPU-intensive

Some tasks are memory-intensive

2000-node Hadoop Cluster at Facebook (Oct 2010)
Modeling Tasks w/ Multiple Resources

- Users have *tasks* according to a *demand vector*
  - e.g. <2, 3, 1> user’s tasks need 2 $R_1$, 3 $R_2$, 1 $R_3$
  - Not needed in practice, can simply measure actual consumption
- Resources given in multiples of demand vectors
- Key issue: how to divide the resources if demands are demand vectors?
First Try: Asset Fairness

- **Asset Fairness**
  - Equalize each user’s sum of resource shares

- **Cluster with 70 CPUs, 70 GB RAM**
  - $U_1$ needs $<2$ CPU, 2 GB RAM per task
  - $U_2$ needs $<1$ CPU, 2 GB RAM per task

- **Asset fairness yields**
  - $U_1$: 15 tasks: 30 CPUs, 30 GB ($\Sigma=60$)
  - $U_2$: 20 tasks: 20 CPUs, 40 GB ($\Sigma=60$)

---

**Problem**

- User 1 has $<50\%$ of both CPUs and RAM

Better off in a separate cluster with $50\%$ of the resources

---

<table>
<thead>
<tr>
<th>CPU</th>
<th>User 1</th>
<th>User 2</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>43%</td>
<td></td>
<td>57%</td>
<td></td>
</tr>
<tr>
<td>28%</td>
<td></td>
<td>57%</td>
<td></td>
</tr>
<tr>
<td>0%</td>
<td>43%</td>
<td>0%</td>
<td></td>
</tr>
</tbody>
</table>

First Try: Asset Fairness

- **Problem**
  - User 1 has $<50\%$ of both CPUs and RAM

Better off in a separate cluster with $50\%$ of the resources

---

<table>
<thead>
<tr>
<th>CPU</th>
<th>User 1</th>
<th>User 2</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>43%</td>
<td></td>
<td>57%</td>
<td></td>
</tr>
<tr>
<td>28%</td>
<td></td>
<td>57%</td>
<td></td>
</tr>
<tr>
<td>0%</td>
<td>43%</td>
<td>0%</td>
<td></td>
</tr>
</tbody>
</table>
Fixing Asset Fairness

“A user shouldn’t do worse than if it runs a smaller, private cluster equal in size to its fair share of each resource”

Thus, given $N$ users, each user should get $\geq \frac{1}{N}$ of her dominating resource (i.e., the resource that she consumes most of)
Dominant Resource Example

- A user’s *dominant resource* is the resource she has the biggest share of.
  - Example:
    - Total resources: `<10 CPU, 4 GB>`
    - User 1’s allocation: `<2 CPU, 1 GB>`
    - Dominant resource is memory as $\frac{1}{4} > \frac{2}{10} = \frac{1}{5}$
Dominant Resource Fairness Allocation Example

- Apply max-min fairness to dominant shares
- Equalize the dominant share of the users

Example:

Total resources: <9 CPU, 18 GB>
User 1 demand: <1 CPU, 4 GB> dominant res: mem
User 2 demand: <3 CPU, 1 GB> dominant res: CPU

```
CPU
(9 total)

6 CPUs

66%

mem
(18 total)

12 GB

2 GB

66%

User 1

3 CPUs

100%

User 2
```
**Algorithm 1 DRF pseudo-code**

\[ R = \langle r_1, \cdots, r_m \rangle \quad \triangleright \text{total resource capacities} \]
\[ C = \langle c_1, \cdots, c_m \rangle \quad \triangleright \text{consumed resources, initially 0} \]
\[ s_i \quad (i = 1..n) \quad \triangleright \text{user } i \text{'s dominant shares, initially 0} \]
\[ U_i = \langle u_{i,1}, \cdots, u_{i,m} \rangle \quad (i = 1..n) \quad \triangleright \text{resources given to user } i, \text{initially 0} \]

**pick** user \( i \) with lowest dominant share \( s_i \)

**D_i** ← demand of user \( i \)'s next task

**if** \( C + D_i \leq R \) **then**

\[ C = C + D_i \quad \triangleright \text{update consumed vector} \]
\[ U_i = U_i + D_i \quad \triangleright \text{update i's allocation vector} \]
\[ s_i = \max_{j=1}^{m} \left\{ \frac{u_{i,j}}{r_j} \right\} \]

**else**

**return** \( \triangleright \text{the cluster is full} \)

**end if**
DRF Fair Properties

- DRF is strategy-proof
- DRF is share guarantee
- DRF is envy-free

See DRF paper for proofs
Evaluation

- Two allocation policies
  - Slot
    - Each machine consists of $k$ slots (e.g. $k=14$)
    - Run at most one task per slot
    - Give jobs “equal” number of slots,
      i.e., apply max-min fairness to slot-count
  - DRF
Utilization of DRF vs Slots

Simulation of Facebook workload

CPU Utilization

Memory Utilization

Time (s)
DRF Adoption and Extension

- **Industry:**
  - Fair scheduler in YARN for multiple resources
  - Mesos (see later)

- **Extensions:**
  - DRFQ: extend to packet processing
  - Choosy: DRF with constraints
  - Hierarchical Scheduling for DRF
DRF Discussion

- What do you like about DRF?

- What are potential issues of DRF?
Efficiency-Fairness Trade-off

- DRF has under-utilized resources

- DRF schedules at the level of tasks (lead to sub-optimal job completion time)

- Fairness is fundamentally at odds with overall efficiency (how to trade-off?)
Architecture So Far

Request: demand + constraints

YARN

Hadoop App  Spark App  ...

response

Key extension complexity?
**Issue of Request/Response**

- Needs a specification mechanism for generic constraints, but this is challenging:
  - Difficult to be robust
  - Hard to anticipate future need
  - Make RM complex

```java
job hello_world = {
    runtime = { cell = "ic" } //what cell should run it in?
    binary = `../hello_world_webserver` //what program to run?
    args = { port = `%port%` }
    requirements = {
        RAM = 100M
        disk = 100M
        CPU = 0.1
    }
    replicas = 10000
}
```
Outline

- Admin and recap
- Cloud data center (CDC) applications/services
  - Fine-grained dataflow programming (e.g., Web apps)
  - Coarse-grained dataflow (e.g., data analytics)
  - Distributed machine learning using parameter server
  - DC resource scheduling
    - overview
    - YARN architecture
    - resource scheduling
    - alternative design: Mesos
Basic Idea

- Resource requests => resource offers
  - Resource manager offers available resources to frameworks/app, let them accept or reject the offer (an example of using reject to achieve its global goal?)
Mesos Architecture

- **MPI job**
  - **MPI scheduler**
  - **Mesos master**
  - **Resource offer**
  - **Mesos slave**
    - **MPI executor**
      - **task**

- **Hadoop job**
  - **Hadoop scheduler**
  - **Mesos master**
  - **Resource offer**
  - **Mesos slave**
    - **MPI executor**
      - **task**

Pick framework to offer resources to
Mesos Architecture

- **MPI job**
  - MPI scheduler
  - Mesos master
  - Mesos slave
  - MPI executor
  - task

- **Hadoop job**
  - Hadoop scheduler
  - Mesos master
  - Mesos slave
  - MPI executor
  - task

**Resource offer =**
list of (node, availableResources)

E.g. \{ (node1, <2 CPUs, 4 GB>),
(nod2e, <3 CPUs, 2 GB>) \}
Mesos Architecture

- **MPI job**
  - MPI scheduler
  - Mesos master
  - Mesos slave
    - MPI executor
    - task

- **Hadoop job**
  - Hadoop scheduler
  - Mesos master
  - Mesos slave
    - Hadoop executor
    - task

---

- **Framework-specific scheduling**
- **Pick framework to offer resources to**
- **Launches and isolates executors**
Let frameworks short-circuit rejection by providing a predicate on resources to be offered:
- E.g. “nodes from list L” or “nodes with > 8 GB RAM”
- Could generalize to other hints as well

Ability to reject still ensures correctness when needs cannot be expressed using filters.
Data Locality with Resource Offers

- Ran 16 instances of Hadoop on a shared HDFS cluster
- Used delay scheduling [EuroSys '10] in Hadoop to get locality (wait a short time to acquire data-local nodes)

![Local Map Tasks (%)](chart1)

![Job Duration (s)](chart2)
Scalability

- Mesos only performs *inter-framework* scheduling (e.g., fair sharing), which is easier than *intra-framework* scheduling.

**Result:**
Scaled to 50,000 emulated slaves, 200 frameworks, 100K tasks (30s len)
Architecture So Far

request: demand + constraints accept/reject
response offer

Hadoop App  Spark App …

YARN

Node  Node  Node  Node
Summary: Existing Resource Allocation Models

Monolithic
- Hard to add new policies
- Scalability bottleneck

Static Partitioning
- Poor utilization
- Inflexible allocation

Two-level
- Limited parallelism (pessimistic)
- Scheduler has limited visibility
Outline

- Admin and recap
- Cloud data center (CDC) applications/services
  - Fine-grained dataflow programming (e.g., Web apps)
  - Coarse-grained dataflow (e.g., data analytics)
  - Distributed machine learning using parameter server
  - DC resource scheduling
    - overview
    - YARN architecture
    - resource scheduling, DRF
    - alternative design: Mesos
    - alternative design: Omega
Omega Basic Idea

- Multiple controllers, shared state
  - Each controller can independently read/write shared state
Omega Basic Idea

- Problem: race condition

![Diagram showing race condition]
Omega Basic Idea

- Rollback if conflict: opportunistic concurrency control

Failed allocation

S1

S2

S3
Discussion

- Benefits of Omega?

- Issues of Omega?